Chapter 4. Using XL builtin floating-point functions for Blue Gene

The XL C/C++ and XL Fortran compilers include a set of built-in functions that are optimized for the PowerPC architecture. For a full description of them, refer to the following documents (available from the Web pages listed at the beginning of this chapter):

- *Built-in functions for POWER™ and PowerPC architectures* in *XL C/C++ Advanced Edition for Linux, V9.0 Compiler Reference*
- *Intrinsic procedures* in *XL Fortran Advanced Edition for Linux, V11.1 Language Reference*

In addition, on Blue Gene, the XL compilers provide a set of built-in functions that are specifically optimized for the PowerPC 440 or PowerPC 450 Double Hummer dual FPU. These built-in functions provide an almost one-to-one correspondence with the Double Hummer instruction set.

All of the C/C++ and Fortran built-in functions operate on complex data types, which have an underlying representation of a two-element array, in which the real part represents the *primary* element and the imaginary part represents the *second* element. The input data you provide does not actually need to represent complex numbers: in fact, both elements are represented internally as two real values, and none of the built-in functions actually performs complex arithmetic. A set of built-in functions especially designed to efficiently manipulate complex-type variables is also available.

The Blue Gene built-in functions perform the several types of operations as explained in the following paragraphs.

Parallel operations perform SIMD computations on the primary and secondary elements of one or more input operands. They store the results in the corresponding elements of the output. As an example, Figure 8 on page 32 illustrates how a parallel multiply operation is performed.

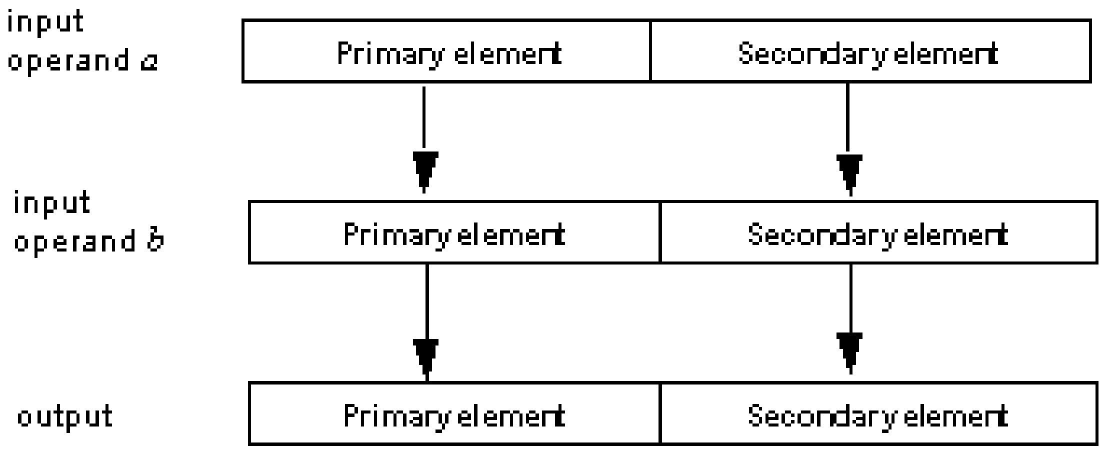

Figure 8. Parallel operations

Cross operations perform SIMD computations on the opposite primary and secondary elements of one or more input operands. They store the results in the corresponding elements in the output. As an example, Figure 9 illustrates how a cross-multiply operation is performed.

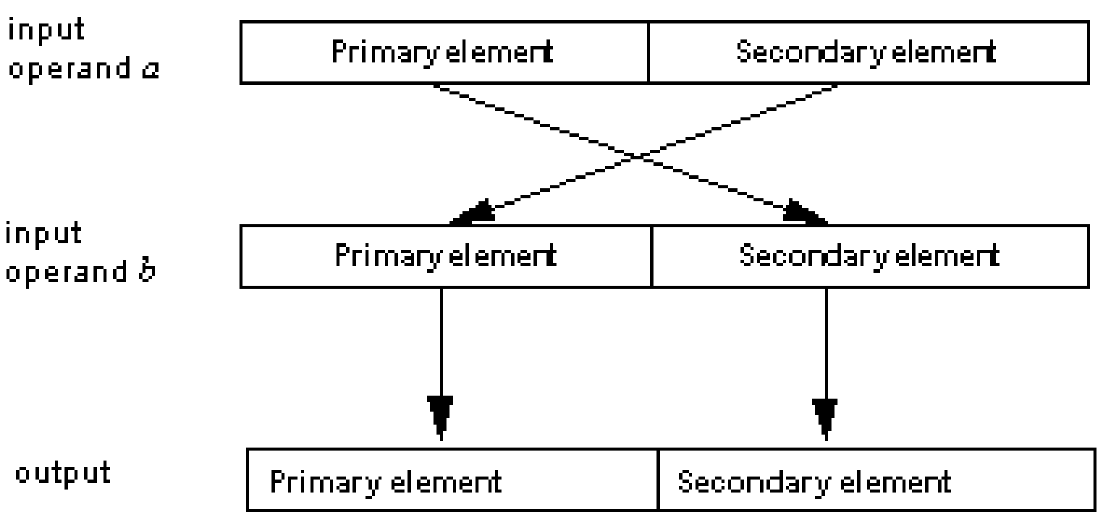

Figure 9. Cross operations

Copy-primary operations perform SIMD computation between the corresponding primary and secondary elements of two input operands, where the primary element of the first operand is replicated to the secondary element. As an example, Figure 10 on page 33 illustrates how a cross-primary multiply operation is performed.

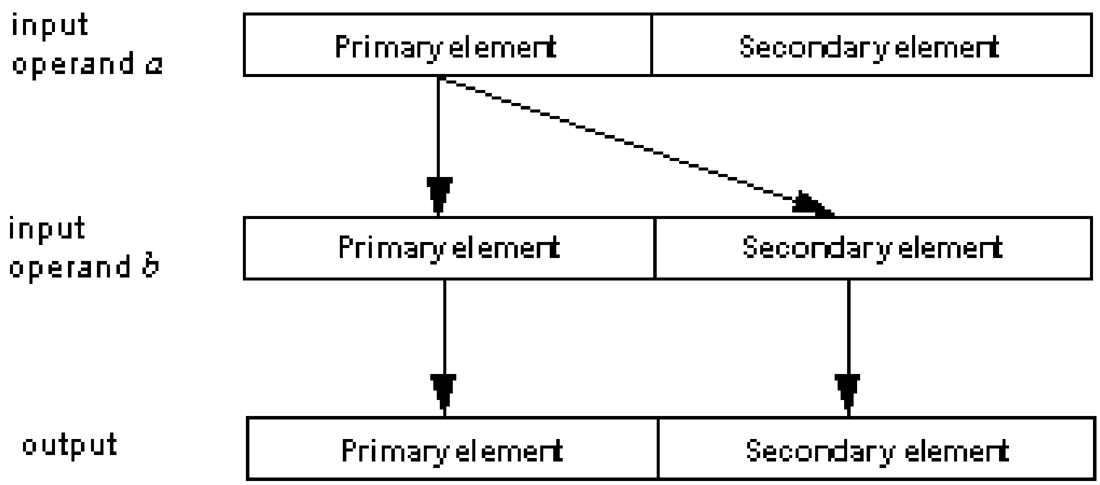

Figure 10. Copy-primary operations

Copy-secondary operations perform SIMD computation between the corresponding primary and secondary elements of two input operands, where the secondary element of the first operand is replicated to the primary element. As an example, Figure 11 illustrates how a cross-secondary multiply operation is performed.

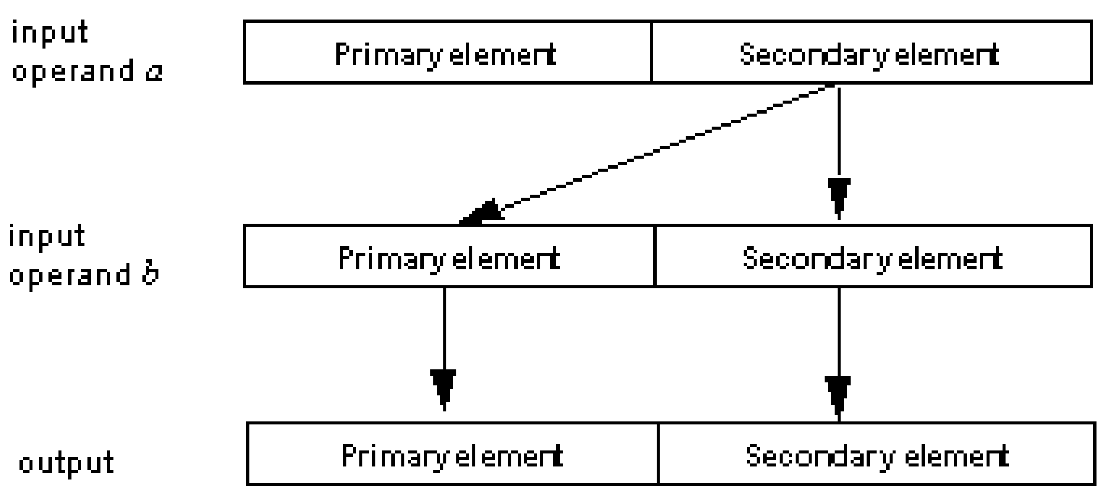

Figure 11. Copy-secondary operations

In *cross-copy* operations, the compiler crosses either the primary or secondary element of the first operand, so that copy-primary and copy-secondary operations can be used interchangeably to achieve the same result. The operation is performed on the total value of the first operand. As an example, Figure 12 on page 34 illustrates the result of a cross-copy multiply operation.

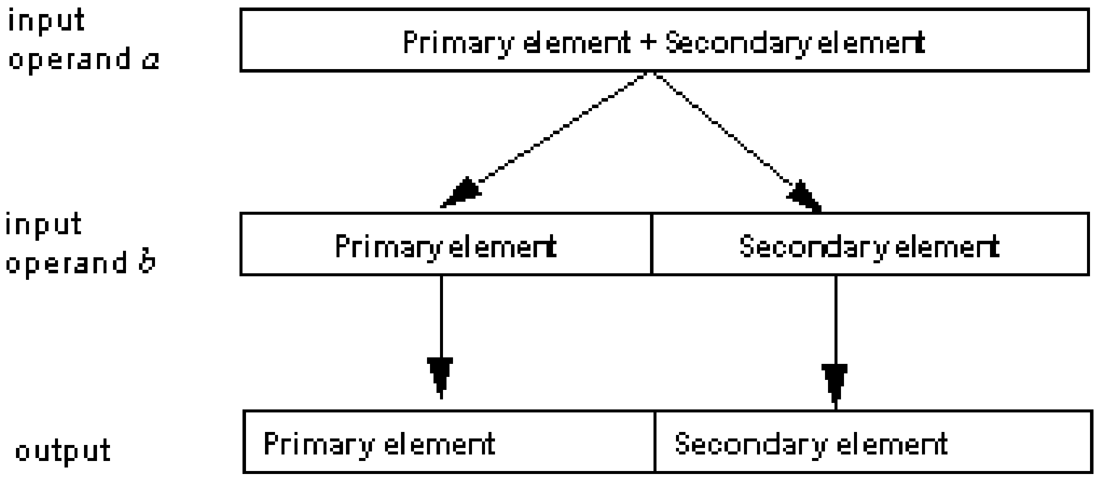

Figure 12. Cross-copy operations

The following sections describe the available built-in functions by category:

- Complex type manipulation functions
- Load and store functions
- Move functions
- Arithmetic functions
- Select functions

For each function, the C/C++ prototype is provided. In C, you do not need to include a header file to obtain the prototypes. The compiler includes them automatically. In C++, you need to include the header file `builtins.h`.

Fortran does not use prototypes for built-in functions. Therefore, the interfaces for the Fortran functions are provided in textual form. The function names omit the double underscore (__) in Fortran.

All of the built-in functions, with the exception of the complex type manipulation functions, require compilation under **-qarch=440d** for Blue Gene/L, or **-qarch=450d** for Blue Gene/P. This is the default setting for these processors.

To help clarify the English description of each function, the following notation is used:

element (*variable*)

where *element* represents one of *primary* or *secondary* , and *variable* represents input variable *a* , *b* , or *c* , and the output variable *result* . For example, consider the following formula:

```
primary(result) = primary(a) + primary(b)
```

The formula indicates that the primary element of input variable *a* is added to the primary element of input variable *b* and stored in the primary element of the *result*.

To optimize your calls to the Blue Gene built-in functions, follow the guidelines provided in Tuning your code for Blue Gene. Using the **alignx** built-in function (described in Checking for data alignment), and specifying the **disjoint** pragma (described in Removing possibilities for aliasing (C/C++)), are recommended for code that calls any of the built-in functions.

Complex type manipulation functions

The functions described in this section are useful for efficiently manipulating complex data types, by allowing you to automatically convert real floating-point data to complex types, and to extract the real (primary) and imaginary (secondary) parts of complex values.

Table 15. Complex type manipulation functions

Function	**Convert dual reals to complex (single-precision): __cmplxf**
Purpose	Converts two single-precision real values to a single complex value. The real *a* is converted to the primary element of the return value, and the real *b* is converted to the secondary element of the return value.
Formula	primary(result) = a secondary(result) = b
C/C++ prototype	float _Complex __cmplxf (float a, float b);
Fortran description	CMPLX(A,B) where A is of type REAL(4) where B is of type REAL(4) result is of type COMPLEX(4)
Function	**Convert dual reals to complex (double-precision): __cmplx**
Purpose	Converts two double-precision real values to a single complex value. The real *a* is converted to the primary element of the return value, and the real *b* is converted to the secondary element of the return value.
Formula	primary(result) = a secondary(result) = b
C/C++ prototype	double _Complex __cmplx (double a, double b); long double _Complex __cmplxl (long double a, long double b);[1]
Fortran description	CMPLX(A,B) where A is of type REAL(8) where B is of type REAL(8) result is of type COMPLEX(8)
Function	**Extract real part of complex (single-precision): __crealf**
Purpose	Extracts the primary part of a single-precision complex value *a* , and returns the result as a single real value.
Formula	result = primary(a)
C/C++ prototype	float __crealf (float _Complex a);
Fortran description	N/A
Function	**Extract real part of complex (double-precision): __creal, __creall**
Purpose	Extracts the primary part of a double-precision complex value *a* , and returns the result as a single real value.

Table 15. Complex type manipulation functions (continued)

Formula	result = primary(a)
C/C++ prototype	double __creal (double _Complex a); long double __creall (long double _Complex a);[1]
Fortran description	N/A
Function	**Extract imaginary part of complex (single-precision): __cimagf**
Purpose	Extracts the secondary part of a single-precision complex value *a* , and returns the result as a single real value.
Formula	result = secondary(a)
C/C++ prototype	float __cimagf (float _Complex a);
Fortran description	N/A
Function	Extract imaginary part of complex (double-precision): __cimag, __cimagl
Purpose	Extracts the imaginary part of a double-precision complex value *a* , and returns the result as a single real value.
Formula	result =secondary(a)
C/C++ prototype	double __cimag (double _Complex a); long double __cimagl (long double _Complex a);[1]
Fortran description	N/A

Notes:

1. 128-bit C/C++ long double types are not supported on Blue Gene/L. Long doubles are treated as regular double-precision doubles.

Load and store functions

Table 16 lists and explains the various parallel load and store functions that are available.

Table 16. Load and store functions

Function	**Parallel load (single-precision): __lfps**
Purpose	Loads parallel single-precision values from the address of *a* , and converts the results to double-precision. The first word in *address(a)* is loaded into the primary element of the return value. The next word, at location *address(a)* +4, is loaded into the secondary element of the return value.
Formula	primary(result) = a[0] secondary(result) = a[1]
C/C++ prototype	double _Complex __lfps (float * a);
Fortran description	LOADFP(A) where A is of type REAL(4) or COMPLEX(4) result is of type COMPLEX(8)
Function	**Cross load (single-precision): __lfxs**

Table 16. Load and store functions (continued)

Purpose	Loads single-precision values that have been converted to double-precision, from the address of *a*. The first word in *address(a)* is loaded into the secondary element of the return value. The next word, at location *address(a)* +4, is loaded into the primary element of the return value.
Formula	primary(result) = a[1] secondary(result) = a[0]
C/C++ prototype	double _Complex __lfxs (float * a);
Fortran description	LOADFX(A) where A is of type REAL(4) or COMPLEX(4) result is of type COMPLEX(8)
Function	**Parallel load: __lfpd**
Purpose	Loads in parallel values from the address of *a*. The first word in *address(a)* is loaded into the primary element of the return value. The next word, at location *address(a)* +8, is loaded into the secondary element of the return value.
Formula	primary(result) = a[0] secondary(result) = a[1]
C/C++ prototype	double _Complex __lfpd(double* a);
Fortran description	LOADFP(A) where A is of type REAL(8) or COMPLEX(8) result is of type COMPLEX(8)
Function	**Cross load: __lfxd**
Purpose	Loads values from the address of *a*. The first word in *address(a)* is loaded into the secondary element of the return value. The next word, at location *address(a)* +8, is loaded into the primary element of the return value.
Formula	primary(result) = a[1] secondary(result) = a[0]
C/C++ prototype	double _Complex __lfxd (double * a);
Fortran description	LOADFX(A) where A is of type REAL(8) or COMPLEX(8) result is of type COMPLEX(8)
Function	**Parallel store (single-precision): __stfps**
Purpose	Stores in parallel double-precision values that have been converted to single-precision, into *address(b)*. The primary element of *a* is converted to single-precision and stored as the first word in *address(b)*. The secondary element of *a* is converted to single-precision and stored as the next word at location *address(b)* +4.
Formula	b[0] = primary(a) b[1] = secondary(a)
C/C++ prototype	void __stfps (float * b, double _Complex a);

Table 16. Load and store functions (continued)

Fortran description	STOREFP(B,A) where B is of type REAL(4) or COMPLEX(4) where A is of type COMPLEX(8) result is none
Function	**Cross store (single-precision): __stfxs**
Purpose	Stores double-precision values that have been converted to single-precision, into *address(b)*. The secondary element of *a* is converted to single-precision and stored as the first word in *address(b)*. The primary element of *a* is converted to single-precision and stored as the next word at location *address(b)* +4.
Formula	b[0] = secondary(a) b[1] = primary(a)
C/C++ prototype	void __stfxs (float * b, double _Complex a);
Fortran description	STOREFX(B,A) where B is of type REAL(4) or COMPLEX(4) where A is of type COMPLEX(8) result is none
Function	**Parallel store: __stfpd**
Purpose	Stores in parallel values into *address(b)*. The primary element of *a* is stored as the first double word in *address(b)*. The secondary element of *a* is stored as the next double word at location *address(b)* +8.
Formula	b[0] = primary(a) b[1] = secondary(a)
C/C++ prototype	void __stfpd (double * b, double _Complex a);
Fortran description	STOREFP(B,A) where B is of type REAL(8) or COMPLEX(8) where A is of type COMPLEX(8) result is none
Function	**Cross store: __stfxd**
Purpose	Stores values into *address(b)*. The secondary element of *a* is stored as the first double word in *address(b)*. The primary element of *a* is stored as the next double word at location *address(b)* +8.
Formula	b[0] = secondary(a) b[1] = primary(a)
C/C++ prototype	void __stfxd (double * b, double _Complex a);
Fortran description	STOREFX(B,A) where B is of type REAL(8) or COMPLEX(8) where A is of type COMPLEX(8) result is none
Function	**Parallel store as integer: __stfpiw**

Table 16. Load and store functions (continued)

Purpose	Stores in parallel floating-point double-precision values into *b* as integer words. The lower-order 32 bits of the primary element of *a* are stored as the first integer word in *address(b)*. The lower-order 32 bits of the secondary element of *a* are stored as the next integer word at location *address(b)* +4. This function is typically preceded by a call to the **__fpctiw** or **__fpctiwz** built-in functions, described in Unary functions, which perform parallel conversion of dual floating-point values to integers.
Formula	b[0] = primary(a) b[1] = secondary(a)
C/C++ prototype	void __stfpiw (int * b, double _Complex a);
Fortran description	STOREFP(B,A) where B is of type INTEGER(4) where A is of type COMPLEX(8) result is none

Move functions

Table 17.

Function	Cross move: __fxmr
Purpose	Swaps the values of the primary and secondary elements of operand *a*.
Formula	primary(result) = secondary(a) secondary(result) = primary(a)
C/C++ prototype	double _Complex __fxmr (double _Complex a);
Fortran description	FXMR(A) where A is of type COMPLEX(8) result is of type COMPLEX(8)

Arithmetic functions

The following sections describe all the arithmetic built-in functions, categorized by their number of operands:

- Unary functions
- Binary functions
- Multiply-add functions

Unary functions

Unary functions operate on a single input operand. These functions are listed in Table 18.

Table 18. Unary functions

Function	Parallel convert to integer: __fpctiw

Table 18. Unary functions (continued)

Purpose	Converts in parallel the primary and secondary elements of operand *a* to 32-bit integers. After a call to this function, use the **__stfpiw** function to store the converted integers in parallel, as described in Load and store functions.
Formula	primary(result) = primary(a) secondary(result) = secondary(a)
C/C++ prototype	double _Complex __fpctiw (double _Complex a);
Fortran description	FPCTIW(A) where A is of type COMPLEX(8) result is of type COMPLEX(8)
Function	**Parallel convert to integer and round to zero: __fpctiwz**
Purpose	Converts in parallel the primary and secondary elements of operand *a* to 32 bit integers and rounds the results to zero. After a call to this function, you will want to use the **__stfpiw** function to store the converted integers in parallel, as described in Load and store functions.
Formula	primary(result) = primary(a) secondary(result) = secondary(a)
C/C++ prototype	double _Complex __fpctiwz(double _Complex a);
Fortran description	FPCTIWZ(A) where A is of type COMPLEX(8) result is of type COMPLEX(8)
Function	**Parallel round double-precision to single-precision: __fprsp**
Purpose	Rounds in parallel the primary and secondary elements of double-precision operand *a* to single precision.
Formula	primary(result) = primary(a) secondary(result) = secondary(a)
C/C++ prototype	double _Complex __fprsp (double _Complex a);
Fortran description	FPRSP(A) where A is of type COMPLEX(8) result is of type COMPLEX(8)
Function	**Parallel reciprocal estimate: __fpre**
Purpose	Calculates in parallel double-precision estimates of the reciprocal of the primary and secondary elements of operand *a*.
Formula	primary(result) = primary(a) secondary(result) = secondary(a)
C/C++ prototype	double _Complex __fpre(double _Complex a);
Fortran description	FPRE(A) where A is of type COMPLEX(8) result is of type COMPLEX(8)
Function	**Parallel reciprocal square root: __fprsqrte**
Purpose	Calculates in parallel double-precision estimates of the reciprocals of the square roots of the primary and secondary elements of operand *a*.

Table 18. Unary functions (continued)

Formula	primary(result) = primary(a) secondary(result) = secondary(a)
C/C++ prototype	double _Complex __fprsqrte (double _Complex a);
Fortran description	FPRSQRTE(A) where A is of type COMPLEX(8) result is of type COMPLEX(8)
Function	**Parallel negate: __fpneg**
Purpose	Calculates in parallel the negative absolute values of the primary and secondary elements of operand *a*.
Formula	primary(result) = primary(a) secondary(result) = secondary(a)
C/C++ prototype	double _Complex __fpneg (double _Complex a);
Fortran description	FPNEG(A) where A is of type COMPLEX(8) result is of type COMPLEX(8)
Function	**Parallel absolute: __fpabs**
Purpose	Calculates in parallel the absolute values of the primary and secondary elements of operand *a*.
Formula	primary(result) = primary(a) secondary(result) = secondary(a)
C/C++ prototype	double _Complex __fpabs (double _Complex a);
Fortran description	FPABS(A) where A is of type COMPLEX(8) result is of type COMPLEX(8)
Function	**Parallel negate absolute: __fpnabs**
Purpose	Calculates in parallel the negative absolute values of the primary and secondary elements of operand *a*.
Formula	primary(result) = primary(a) secondary(result) = secondary(a)
C/C++ prototype	double _Complex __fpnabs (double _Complex a);
Fortran description	FPNABS(A) where A is of type COMPLEX(8) result is of type COMPLEX(8)

Binary functions

Binary functions operate on two input operands. The functions are listed in Table 19.

Table 19.

Function	**Parallel add: __fpadd**

Table 19. (continued)

Purpose	Adds in parallel the primary and secondary elements of operands *a* and *b*.
Formula	primary(result) = primary(a) + primary(b) secondary(result) = secondary(a) + secondary(b)
C/C++ prototype	double _Complex __fpadd (double _Complex a, double _Complex b);
Fortran description	FPADD(A,B) where A is of type COMPLEX(8) where B is of type COMPLEX(8) result is of type COMPLEX(8)
Function	**Parallel subtract: __fpsub**
Purpose	Subtracts in parallel the primary and secondary elements of operand *b* from the corresponding primary and secondary elements of operand *a*.
Formula	primary(result) = primary(a) - primary(b) secondary(result) = secondary(a) - secondary(b)
C/C++ prototype	double _Complex __fpsub (double _Complex a, double _Complex b);
Fortran description	FPSUB(A,B) where A is of type COMPLEX(8) where B is of type COMPLEX(8) result is of type COMPLEX(8)
Function	**Parallel multiply: __fpmul**
Purpose	Multiples in parallel the values of primary and secondary elements of operands *a* and *b*.
Formula	primary(result) = primary(a) × primary(b) secondary(result) = secondary(a) × secondary(b)
C/C++ prototype	double _Complex __fpmul (double _Complex a, double _Complex b);
Fortran description	FPMUL(A,B) where A is of type COMPLEX(8) where B is of type COMPLEX(8) result is of type COMPLEX(8)
Function	Cross multiply: __fxmul
Purpose	The product of the secondary element of *a* and the primary element of *b* is stored as the primary element of the return value. The product of the primary element of *a* and the secondary element of *b* is stored as the secondary element of the return value.
Formula	primary(result) = secondary(a) x primary(b) secondary(result) = primary(a) × secondary(b)
C/C++ prototype	double _Complex __fxmul (double _Complex a, double _Complex b);
Fortran description	FXMUL(A,B) where A is of type COMPLEX(8) where B is of type COMPLEX(8) result is of type COMPLEX(8)
Function	**Cross copy multiply: _fxpmul, __fxsmul**

Table 19. (continued)

Purpose	Both of these functions can be used to achieve the same result. The product of *a* and the primary element of *b* is stored as the primary element of the return value. The product of *a* and the secondary element of *b* is stored as the secondary element of the return value.
Formula	primary(result) = a x primary(b) secondary(result) = a x secondary(b)
C/C++ prototype	double _Complex __fxpmul (double _Complex b, double a); double _Complex __fxsmul (double _Complex b, double a);
Fortran description	FXPMUL(B,A) or FXSMUL(B,A) where B is of type COMPLEX(8) where A is of type COMPLEX(8) result is of type COMPLEX(8)

Multiply-add functions

Multiply-add functions take three input operands, multiply the first two, and add or subtract the third.

Table 20.

Function	Parallel multiply-add: __fpmadd
Purpose	The sum of the product of the primary elements of *a* and *b*, added to the primary element of *c*, is stored as the primary element of the return value. The sum of the product of the secondary elements of *a* and *b*, added to the secondary element of *c*, is stored as the secondary element of the return value.
Formula	primary(result) = primary(a) × primary(b) + primary(c) secondary(result) = secondary(a) × secondary(b) + secondary(c)
C/C++ prototype	double _Complex __fpmadd (double _Complex c, double _Complex b, double _Complex a);
Fortran description	FPMADD(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type COMPLEX(8) result is of type COMPLEX(8)
Function	**Parallel negative multiply-add: __fpnmadd**
Purpose	The sum of the product of the primary elements of *a* and *b*, added to the primary element of *c*, is negated and stored as the primary element of the return value. The sum of the product of the secondary elements of *a* and *b*, added to the secondary element of *c*, is negated and stored as the secondary element of the return value.
Formula	primary(result) = -(primary(a) × primary(b) + primary(c)) secondary(result) = -(secondary(a) × secondary(b) + secondary(c))
C/C++ prototype	double _Complex __fpnmadd (double _Complex c, double _Complex b, double _Complex a);
Fortran description	FPNMADD(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type COMPLEX(8) result is of type COMPLEX(8)

Table 20. (continued)

Function	Parallel multiply-subtract: __fpmsub
Purpose	The difference of the primary element of *c*, subtracted from the product of the primary elements of *a* and *b*, is stored as the primary element of the return value. The difference of the secondary element of *c*, subtracted from the product of the secondary elements of *a* and *b*, is stored as the secondary element of the return value.
Formula	primary(result) = primary(a) × primary(b) - primary(c) secondary(result) = secondary(a) × secondary(b) - secondary(c)
C/C++ prototype	double _Complex __fpmsub (double _Complex c, double _Complex b, double _Complex a);
Fortran description	FPMSUB(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type COMPLEX(8) result is of type COMPLEX(8)
Function	**Parallel negative multiply-subtract: __fpnmsub**
Purpose	The difference of the primary element of *c*, subtracted from the product of the primary elements of *a* and *b*, is negated and stored as the primary element of the return value. The difference of the secondary element of *c*, subtracted from the product of the secondary elements of *a* and *b*, is negated and stored as the secondary element of the return value.
Formula	primary(result) = -(primary(a) × primary(b) - primary(c)) secondary(result) = -(secondary(a) × secondary(b) - secondary(c))
C/C++ prototype	double _Complex __fpnmsub (double _Complex c, double _Complex b, double _Complex a);
Fortran description	FPNMSUB(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type COMPLEX(8) result is of type COMPLEX(8)
Function	**Cross multiply-add: __fxmadd**
Purpose	The sum of the product of the primary element of *a* and the secondary element of *b*, added to the primary element of *c*, is stored as the primary element of the return value. The sum of the product of the secondary element of *a* and the primary element of *b*, added to the secondary element of *c*, is stored as the secondary element of the return value.
Formula	primary(result) = primary(a) × secondary(b) + primary(c) secondary(result) = secondary(a) × primary(b) + secondary(c)
C/C++ prototype	double _Complex __fxmadd (double _Complex c, double _Complex b, double _Complex a);
Fortran description	FXMADD(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type COMPLEX(8) result is of type COMPLEX(8)
Function	**Cross negative multiply-add: __fxnmadd**

Table 20. (continued)

Purpose	The sum of the product of the primary element of *a* and the secondary element of *b*, added to the primary element of *c*, is negated and stored as the primary element of the return value. The sum of the product of the secondary element of *a* and the primary element of *b*, added to the secondary element of *c*, is negated and stored as the secondary element of the return value.
Formula	primary(result) = -(primary(a) × secondary(b) + primary(c)) secondary(result) = -(secondary(a) × primary(b) + secondary(c))
C/C++ prototype	double _Complex __fxnmadd (double _Complex c, double _Complex b, double _Complex a);
Fortran description	FXNMADD(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type COMPLEX(8) result is of type COMPLEX(8)
Function	**Cross multiply-subtract: __fxmsub**
Purpose	The difference of the primary element of *c*, subtracted from the product of the primary element of *a* and the secondary element of *b*, is stored as the primary element of the return value. The difference of the secondary element of *c*, subtracted from the product of the secondary element of *a* and the primary element of *b*, is stored as the secondary element of the return value.
Formula	primary(result) = primary(a) × secondary(b) - primary(c) secondary(result) = secondary(a) × primary(b) - secondary(c)
C/C++ prototype	double _Complex __fxmsub (double _Complex c, double _Complex b, double _Complex a);
Fortran description	FXMSUB(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type COMPLEX(8) result is of type COMPLEX(8)
Function	**Cross negative multiply-subtract: __fxnmsub**
Purpose	The difference of the primary element of *c*, subtracted from the product of the primary element of *a* and the secondary element of *b*, is negated and stored as the primary element of the return value. The difference of the secondary element of *c*, subtracted from the product of the secondary element of *a* and the primary element of *b*, is negated and stored as the secondary element of the return value.
Formula	primary(result) = -(primary(a) × secondary(b) - primary(c)) secondary(result) = -(secondary(a) × primary(b) - secondary(c))
C/C++ prototype	double _Complex __fxnmsub (double _Complex c, double _Complex b, double _Complex a);
Fortran description	FXNMSUB(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type COMPLEX(8) result is of type COMPLEX(8)
Function	**Cross copy multiply-add: __fxcpmadd, __fxcsmadd**

Table 20. (continued)

Purpose	Both of these functions can be used to achieve the same result. The sum of the product of *a* and the primary element of *b*, added to the primary element of *c* , is stored as the primary element of the return value. The sum of the product of *a* and the secondary element of *b*, added to the secondary element of *c* , is stored as the secondary element of the return value.
Formula	primary(result) = a x primary(b) + primary(c) secondary(result) = a x secondary(b) + secondary(c)
C/C++ prototype	double _Complex __fxcpmadd (double _Complex c, double _Complex b, double a); double _Complex __fxcsmadd (double _Complex c, double _Complex b, double a);
Fortran description	FXCPMADD(C,B,A) or FXCSMADD(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type REAL(8) result is of type COMPLEX(8)
Function	**Cross copy negative multiply-add: __fxcpnmadd, __fxcsnmadd**
Purpose	Both of these functions can be used to achieve the same result. The difference of the primary element of *c*, subtracted from the product of *a* and the primary element of *b*, is negated and stored as the primary element of the return value. The difference of the secondary element of *c* , subtracted from the product of *a* and the secondary element of *b*, is negated stored as the secondary element of the return value.
Formula	primary(result) = -(a x primary(b) + primary(c)) secondary(result) = -(a x secondary(b) + secondary(c))
C/C++ prototype	double _Complex __fxcpnmadd (double _Complex c, double _Complex b, double a); double _Complex __fxcsnmadd (double _Complex c, double _Complex b, double a);
Fortran description	FXCPNMADD(C,B,A) or FXCSNMADD(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type REAL(8) result is of type COMPLEX(8)
Function	**Cross copy multiply-subtract: __fxcpmsub, __fxcsmsub**
Purpose	Both of these functions can be used to achieve the same result. The difference of the primary element of *c*, subtracted from the product of *a* and the primary element of *b*, is stored as the primary element of the return value. The difference of the secondary element of *c*, subtracted from the product of *a* and the secondary element of *b*, is stored as the secondary element of the return value.
Formula	primary(result) = a x primary(b) - primary(c) secondary(result) = a x secondary(b) - secondary(c)
C/C++ prototype	double _Complex __fxcpmsub (double _Complex c, double _Complex b, double a); double _Complex __fxcsmsub (double _Complex c, double _Complex b, double a);

Table 20. (continued)

Fortran description	FXCPMSUB(C,B,A) or FXCSMSUB(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type REAL(8) result is of type COMPLEX(8)
Function	**Cross copy negative multiply-subtract: __fxcpnmsub, __fxcsnmsub**
Purpose	Both of these functions can be used to achieve the same result. The difference of the primary element of *c*, subtracted from the product of *a* and the primary element of *b*, is negated and stored as the primary element of the return value. The difference of the secondary element of *c*, subtracted from the product of *a* and the secondary element of *b*, is negated stored as the secondary element of the return value.
Formula	primary(result) = -(a x primary(b) - primary(c)) secondary(result) = -(a x secondary(b) - secondary(c))
C/C++ prototype	double _Complex __fxcpnmsub (double _Complex c, double _Complex b, double a); double _Complex __fxcsnmsub (double _Complex c, double _Complex b, double a);
Fortran description	FXCPNMSUB(C,B,A) or FXCSNMSUB(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type REAL(8) result is of type COMPLEX(8)
Function	**Cross copy sub-primary multiply-add: __fxcpnpma, __fxcsnpma**
Purpose	Both of these functions can be used to achieve the same result. The difference of the primary element of *c*, subtracted from the product of *a* and the primary element of *b*, is negated and stored as the primary element of the return value. The sum of the product of *a* and the secondary element of *b*, added to the secondary element of *c*, is stored as the secondary element of the return value.
Formula	primary(result) = -(a x primary(b) - primary(c)) secondary(result) = a x secondary(b) + secondary(c)
C/C++ prototype	double _Complex __fxcpnpma (double _Complex c, double _Complex b, double a); double _Complex __fxcsnpma (double _Complex c, double _Complex b, double a);
Fortran description	FXCPNPMA(C,B,A) or FXCSNPMA(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type REAL(8) result is of type COMPLEX(8)
Function	**Cross copy sub-secondary multiply-add: __fxcpnsma, __fxcsnsma**
Purpose	Both of these functions can be used to achieve the same result. The sum of the product of *a* and the primary element of *b*, added to the primary element of *c*, is stored as the primary element of the return value. The difference of the secondary element of *c*, subtracted from the product of *a* and the secondary element of *b*, is negated and stored as the secondary element of the return value.
Formula	primary(result) = a x primary(b) + primary(c)) secondary(result) = -(a x secondary(b) - secondary(c))

Table 20. (continued)

C/C++ prototype	double _Complex ____fxcpnsma (double _Complex c, double _Complex b, double a); double _Complex __fxcsnsma (double _Complex c, double _Complex b, double a);
Fortran description	FXCPNSMA(C,B,A) or FXCSNSMA(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type REAL(8) result is of type COMPLEX(8)
Function	**Cross mixed multiply-add: __fxcxma**
Purpose	The sum of the product of *a* and the secondary element of *b*, added to the primary element of *c*, is stored as the primary element of the return value. The sum of the product of *a* and the primary element of *b*, added to the secondary element of *c*, is stored as the secondary element of the return value.
Formula	primary(result) = a x secondary(b) + primary(c) secondary(result) = a x primary(b) +secondary(c)
C/C++ prototype	double _Complex __fxcxma (double _Complex c, double _Complex b, double a);
Fortran description	FXCXMA(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type REAL(8) result is of type COMPLEX(8)
Function	**Cross mixed negative multiply-subtract: __fxcxnms**
Purpose	The difference of the primary element of *c*, subtracted from the product of *a* and the secondary element of *b*, is negated and stored as the primary element of the return value. The difference of the secondary element of *c*, subtracted from the product of *a* and the primary element of *b*, is negated and stored as the primary secondary of the return value.
Formula	primary(result) = -(a × secondary(b) - primary(c)) secondary(result) = -(a × primary(b) - secondary(c))
C/C++ prototype	double _Complex __fxcxnms (double _Complex c, double _Complex b, double a);
Fortran description	FXCXNMS(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type REAL(8) result is of type COMPLEX(8)
Function	**Cross mixed negative sub-primary multiply-add: __fxcxnpma**
Purpose	The difference of the primary element of *c*, subtracted from the product of the secondary element of *a* and the secondary element of *b*, is negated and stored as the primary element of the return value. The sum of the product of *a* and the primary element of *b*, added to the secondary element of *c*, is stored as the secondary element of the return value.
Formula	primary(result) = -(secondary(a) × secondary(b) - primary(c)) secondary(result) = a × primary(b) + secondary(c)

Table 20. (continued)

C/C++ prototype	double _Complex __fxcxnpma (double _Complex c, double _Complex b, double a);
Fortran description	FXCXNPMA(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type REAL(8) result is of type COMPLEX(8)
Function	**Cross mixed sub-secondary multiply-add: __fxcxnsma**
Purpose	The sum of the product of *a* and the secondary element of *b*, added to the primary element of *c*, is stored as the primary element of the return value. The difference of the secondary element of *c*, subtracted from the product of *a* and the primary element of *b*, is stored as the secondary element of the return value.
Formula	primary(result) = a x secondary(b) + primary(c)) secondary(result) = -(a x primary(b) - secondary(c))
C/C++ prototype	double _Complex __fxcxnsma (double _Complex c, double _Complex b, double a);
Fortran description	FXCXNSMA(C,B,A) where C is of type COMPLEX(8) where B is of type COMPLEX(8) where A is of type REAL(8) result is of type COMPLEX(8)

Select functions

Table 21 lists and explains the select functions that are available.

Table 21. Select functions

Function	**Parallel select: __fpsel**
Purpose	The value of the primary element of *a* is compared to zero. If its value is equal to or greater than zero, the primary element of *c* is stored in the primary element of the return value. Otherwise, the primary element of *b* is stored in the primary element of the return value. The value of the secondary element of *a* is compared to zero. If its value is equal to or greater than zero, the secondary element of *c* is stored in the secondary element of the return value. Otherwise, the secondary element of *b* is stored in the secondary element of the return value.
Formula	primary(result) = if primary(a) ≥ 0 then primary(c); else primary(b) secondary(result) = if secondary(a) ≥ 0 then primary(c); else secondary(b)
C/C++ prototype	double _Complex __fpsel (double _Complex a, double _Complex b, double _Complex c);
Fortran description	FPSEL(A,B,C) where A is of type COMPLEX(8) where B is of type COMPLEX(8) where C is of type COMPLEX(8) result is of type COMPLEX(8)